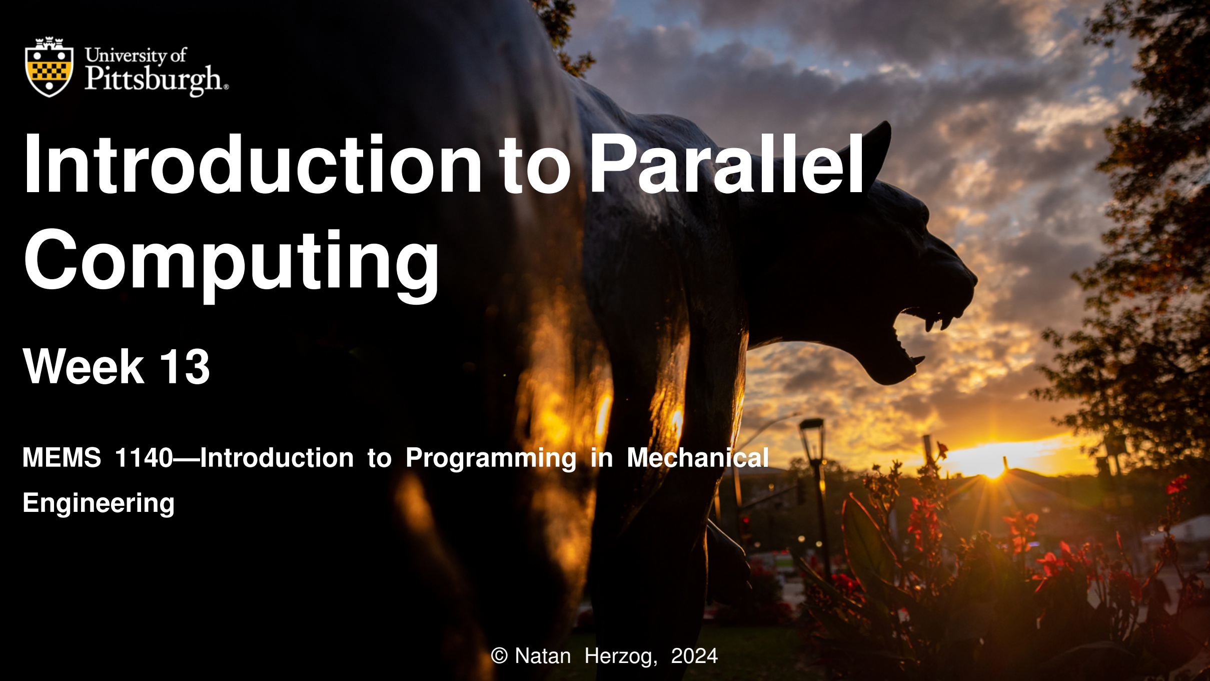


University of Pittsburgh

Introduction to Parallel Computing

Week 13

MEMS 1140—Introduction to Programming in Mechanical Engineering

Learning Objectives (L.O.)

At the end of this lecture, you should understand/be able to:

- ❑ What parallel computing is;
- ❑ How parallelization is useful;
- ❑ Write parallelized loops;
- ❑ Limitations of parallel computing in MATLAB.

Table of Contents (ToC)

1 – Foundation

Up to this point in the semester, we have been writing serialized code, where:

Each operation is processed in *series*, meaning one at a time.

Parallelized code executes multiple operations simultaneously by taking advantage of how computer processors are built.

Let’s briefly explore this architecture.

1 – The CPU

The Central Processing Unit (CPU) is your computer's brain.

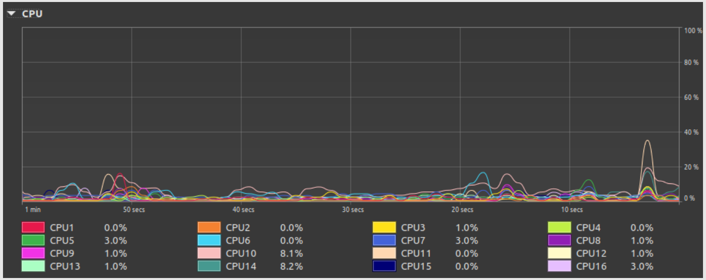
Every operation that happens on your computer passes through the CPU, even if it isn't actually *calculated* there.

It handles: arithmetic and logic; managing memory; planning execution order for processes; etc.

Modern computer processors contain several of these CPUs, commonly called "cores." Each core operates independently.

1 – Computer Processor Activity

At any given moment, your CPU cores are processing tons of operations to keep your machine running. My CPU activity:



🟥	CPU1	0.0%	🟧	CPU2	0.0%	🟨	CPU3	1.0%	🟩	CPU4	0.0%
🟩	CPU5	3.0%	🟦	CPU6	0.0%	🟦	CPU7	3.0%	🟪	CPU8	1.0%
🟪	CPU9	1.0%	🟥	CPU10	8.1%	🟧	CPU11	0.0%	🟨	CPU12	1.0%
🟩	CPU13	1.0%	🟩	CPU14	8.2%	🟦	CPU15	0.0%	🟪	CPU16	3.0%

2 – Serialized vs. Parallelized Code

The code we’ve written this semester all utilized just one core.

We can accelerate our code by distributing it across more than one core, thereby parallelizing its execution.

By doing this, we can reduce our code’s execution time by a factor approximately equal to the number of cores being used.

This enables *really* large studies that would otherwise take far too long. This is also the foundation of supercomputing.

3 – Writing a Serialized `for` Loop

Recall that a regular `for` loop is written as follows:

```
fprintf('serialized:\t')
for i = 1 : 8
  fprintf('| %d | ', i)
end
fprintf('\n')
```

which produces the following output:

```
serialized: | 1 | | 2 | | 3 | | 4 | | 5 | | 6 | | 7 | | 8 |
```

Command Window Output

3 – Writing a Parallelized parfor Loop

The parfor keyword initiates a parallelized loop instead:

```
fprintf('parallel:\t')
parfor i = 1 : 8
  fprintf('| %d | ', i)
end
fprintf('\n')
```

```
Starting parallel pool (parpool) using the 'Processes' profile ...
Connected to parallel pool with 8 workers.
parallel:    | 2 | | 5 | | 4 | | 7 | | 3 | | 1 | | 6 | | 8 |
```

Command Window Output

3 – Takeaway

The serialized version performs exactly as we expect it to, printing out each incrementing number from 1 to 8.

But the parallelized version does not.

This is because the processor distributes the iterations of the loop across the available workers, 8 of them in this case.

All 8 cores print their text at (roughly) the same time. In reality, only one core is given access to write its output at a time.

University of Pittsburgh

PITT | SWANSON ENGINEERING
MECHANICAL & MATERIALS SCIENCE

3 – Inconsistent Execution Order

✓ L.O.1
✓ L.O.2
➩ L.O.3
❏ L.O.4

Even more interesting, the printing order is inconsistent with each subsequent run.

```
parallel pool 1:  | 3 | | 4 | | 2 | | 7 | | 6 | | 1 | | 8 | | 5 |
parallel pool 2:  | 3 | | 7 | | 1 | | 4 | | 6 | | 8 | | 5 | | 2 |
parallel pool 3:  | 5 | | 6 | | 1 | | 4 | | 2 | | 8 | | 7 | | 3 |
```

Command Window Output

Here we see that the workload is distributed and executed differently with each new parallel pool.

3 – Updating the Bridge Study

Let's return to the parametric sweep from Lecture 10.

Let's add a timer and use the `parfor` keyword (next page):

```
Length of thetas array: 89
Serialized Loop time: 1.2378 [s]
Parallelized Loop time: 1.3515 [s]
```

Command Window Output

Oddly enough, the parallelized version is *slower*? What if we increase the problem size?

3 – Updated Code

```
p = parpool;
syms RAx RAy REy AB AC BC BD CD CE DE `applied_load
rhs = @(node_load) [0;0;0;0;0;node_load;0;0;0;0];
loop_start = tic;
parfor i = 1 : length(theta_values)
  theta = theta_values(i);
  % <----- Method of Joints symbolic equations omitted for space ----->
  eqns = [Ax Ay Bx By Cx Cy Dx Dy Ex Ey];
  coeffs = double(equationsToMatrix(eqns));
  [load_capacity, performance] = evaluateBridge(0,theta,...
                                     2886,1755,coeffs(:,1:end-1),rhs);
end
fprintf('Parallelized Loop time: %.4f [s]\n', toc(loop_start))
```

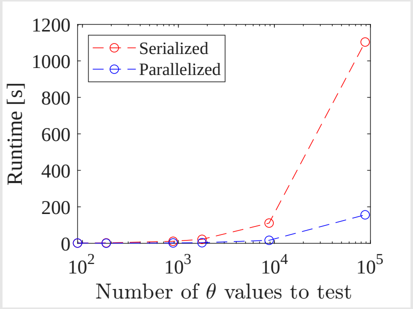
3 – Increasing the Problem Size

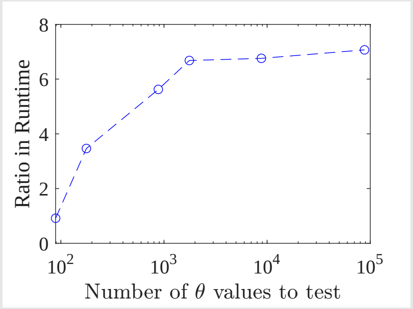
Let's repeat this test with larger problem sizes:

N_θ	Serialized Runtime [s]	Parallelized Runtime [s]	Ratio
89	1.2378	1.3515	0.9159
177	2.1931	0.6330	3.4646
881	11.0640	1.9676	5.6231
1761	21.6858	3.2464	6.6800
8801	110.8977	16.4094	6.7582
88001	1103.2176	156.0790	7.0683

3 – Runtime Graph

To really visualize the power of parallelization, we can plot the results from the previous page (See Amdahl's law):





3 – Power of Parallelization

The power of parallelized code stems from extremely large problems like parametric sweep studies.

As we have seen, the effects of parallelized code become more pronounced as the problem size increases.

We’ve seen a hint of this already, but at this point you might be wondering if there are any *downsides*.

Yes.

4 – Initialization Overhead

MATLAB has to initialize the “pool” of “workers” before the cores can execute any code, about 12 [s] on my computer.

I have completely neglected to tell you about this part so far.

For small problems, initialization alone likely takes longer than a serialized implementation of the same code.

As the problem increases in scale, this initialization becomes negligible. Compare 12 [s] to 1103 [s] from our example.

4 – Parallelization Overhead

In addition to initialization, there is also overhead for managing the parallel processes.

MATLAB has to decide how to segment the operations and manage each core's access to memory, among other things.

These things require some compute overhead, which is much more noticeable for small problems.

Recall that runtime *decreased* from $N_\theta = 89 \rightarrow 177$.

4 – Loop Iteration Independence

For parallelization of loops like we see here, iterations cannot be interdependent.

Each core simultaneously executes different iterations.

The code will fail if iteration **`i=10`** depends on data from iteration **`i=9`**.

With that said, there is a whole field dedicated to solving this. If you're interested, consider taking a distributed systems course!

5 – Summary

✓ L.O.1
✓ L.O.2
✓ L.O.3
✓ L.O.4

This lecture covered:

✓ What parallel computing is

Executing code using multiple processor cores.

✓ How parallelization is useful

Large problems can be dramatically accelerated by distributing the execution across multiple cores.

5 – Summary

✓ L.O.1
✓ L.O.2
✓ L.O.3
✓ L.O.4

✓ How to write parallelized loops

The `parfor` keyword transforms a regular loop into one that can utilize multiple processor cores.

✓ Some limitations of parallel computing

There is considerable overhead in MATLAB from both initializing the parallel pool and managing the cores' execution process. Additionally, loop iterations cannot be interdependent, or the parallelized code will fail.